11 Потоки, управление потоками и взаимодействие потоков

11.1 Концепция вычислительных потоков

Вычислительный поток (***Thread***) – ветвь алгоритма, последовательность («поток») машинных команд, выполняемая параллельно и независимо относительно других таких же последовательностей.

Два основных подхода:

Поток и процесс – две самостоятельные сущности: процесс – владелец ресурсов и контейнер потоков, поток – получатель времени процессора, участник планирования времени выполнения (Win 32/64). Потоки одного процесса естественным образом разделяют и совместно используют его ресурсы, включая адресное пространство.

Поток – «облегченный» процесс (большинство Unix-систем). Группа облегченных процессов совместно использует общий ресурсов, включая совместный доступ к адресному пространству.

11.2 Потоки в Unix-системах

11.2.1 Потоки POSIX Thread – pthread

Первая реализация в Linux (Linux Thread) – libpthread в glibc 2.0, 1996 г.

Дальнейшее развитие – в основном две группы, занимающиеся разработкой на основе разных моделей многопоточности:

NPTL (***N***ative ***P***OSIX ***T***hread ***L***ibrary) – модель потоков ***1:1***

NGPT (***N***ext ***G***eneration ***P***OSIX ***T***hread) – модель потоков ***M:N***

Более простая модель NPTL оказалась и более успешной.

2002 г. – фактическое объединение групп, стандарт – NPTL (с использованием отдельных элементов NGPT)

Общее концептуальное решение:

– управление временем выполнения программ – на уровне процессов, не вводя новых сущностей

– роль потоков исполняют «облегченные» процессы с пересекающимися пространствами адресов памяти и дескрипторов объектов

– роль процессов исполняют ***группы*** потоков

Таким образом, происходит смещение понятий:

$$\text{процесс} \rightarrow \text{поток}$$

$$\text{группа процессов} \rightarrow \text{процесс}$$

Идентификация потока:
– целочисленный ID – Thread ID, ***TID***
– тип **`pthread_`**t – в программах

Значения TID выбираются из того же пространства, что и PID (фактически для системы все PID – подмножество всех TID.
Для головного «потока процесса» (группы) значения PID и TID совпадают. Точнее, в качестве PID процесса (группы) выступает TID головного потока.
Для всех остальных потоков процесса (группы) значение PID такое же, как и для головного, а значения TID индивидуальные для каждого из потоков.

Продолжает действовать системный вызов **`getpid()`**, но добавляется также вызов **`gettid()`**.

(Во многих версиях SDK по каким-то причинам нет функции-«обертки» для этого системного вызова. Воспользоваться им можно с помощью универсальной функции обращения к произвольному системному вызову: **`syscall( SYS_gettid)`**; здесь **`SYS_gettid`** – символическая константа, номер системного вызова, она в SDK как правило, присутствует.)

При компиляции многопоточной программы потребуется опция gcc **`-pthread`**
Заголовочные файлы – **`<pthread.h>`**, **`<unistd.h>`** и др.

Согласно спецификациям POSIX, большинство функций (системных вызовов) в современных версиях Unix/Linux являются в «стандартной» реализации потокобезопасными. В спецификациях оговаривается и список исключений.

11.2.2 Порождение и завершение потоков

Поток создается из ***процедуры*** (***функции***) ***потока***:

```
void* MyThreadRoutine( void* pArg) { … }
```

Создание нового потока:

```
pthread_create( pthread_t *pNewThread, const
pthread_attr_t *pAttr,
            void *(*ThreadRoutine)(void *), void
*pArg);
```

Атрибуты потока – тип данных **pthread_attr_t**, вместо прямого обращения к содержимому используются функции/макросы доступа.

Некоторые из атрибутов: ...

В типичном простейшем случае атрибуты при создании потока могут быть опущены (в функцию **`pthread_create()`** передается пустой указатель).

Обычное корректное завершение текущего потока:

```
void pthread_exit(void *retval);
```

Принудительное завершение произвольного потока (в пределах прав доступа):

```
int pthread_cancel(pthread_t thread);
```

Традиционная функция **`exit()`** (и добавленная в API **`exit_group()`**) завершает все потоки одного «процесса» (группы). Вызвать их может любой из потоков, не обязательно главный.

Состояние «зомби» завершившегося потока зависит от его типа (см. ниже):

– joinable – остаются в состоянии «зомби» до «присоединения» (`join()`) другим потоком

– detached – удаляются сразу после завершения

Принудительное завершение потоков реализовано посредством сигналов – первый по счету сигнал «реального времени», т.е. **32**. В силу этой особенности завершение не выполняется мгновенно. Поведение потока управляется вызовами **`pthread_setcanceltype()`** и **`pthread_setcanceltype()`**:

– ***deferred*** – завершение задерживается до ближайшей т.н. «cancellation points»

– ***asynchronous*** – завершение может происходить в произвольный момент времени, но не обязательно немедленно

– ***disabled*** – принудительное завершение запрещено, исполнение запроса откладывается до момента его разрешения.

В роли «cancellation points» выступает ряд системных вызовов, в основном связанных с вводом-выводом или иным ожиданием; в т.ч. **`pthread_testcancel()`**.

11.2.3 Синхронизация потоков – join. Потоки joinable и detached

Тип потока – joinable или detached – управляется атрибутом (при создании) или вызовом **`pthread_detach()`**.

Joinable – после завершения остается в состоянии «зомби» пока не будет запроса синхронизации с этим событием от другого потока. Этому потоку станут доступны код результата и другая информация о завершившемся.

Detached («отсоединенный») – выполняется полностью самостоятельно (хотя и в рамках группы), после завершения удаляется из системы и не может участвовать в синхронизации.

Ожидание завершения заданного joinable потока:

```
int pthread_join(pthread_t thread, void **retval);
```

(Аналог в Win API – функция **`WaitFor***()`**, которой в качестве аргумента передается handle интересующего потока.)

10.2.4 Сигналы в многопоточном приложении

Потоки сохраняют полную способность генерировать, получать и обрабатывать сигналы Unix.

Отсылка сигнала конкретному потоку в «процессе» (группе):

```
int tkill( int tid, int sig);
int tgkill( int tgid, int tid, int sig);
```

Подобно **`gettid()`**, они не имеют в SDK готовых «оберточных» функций.

Сигнал, отправленный классическим вызов **`kill()`**, адресуется «процессу» в целом – он будет доставлен одному из потоков группы, выбранному произвольно.

11.3 Средства синхронизации потоков

Потокам остаются доступны традиционные средства синхронизации и обмена данными: каналы, сокеты, System V IPC, POSIX IPC и т.д., но их использование сопровождается дополнительными затратами, необходимыми для организации взаимодействия между изолированными друг от друга процессами. Особенность потоков – разделяемое адресное пространство, что позволяет эффективно обмениваться данными через общую память и создавать объекты синхронизации в пользовательской части адресного пространства (с меньшими затратами).

В то же время, потоки более экономны по сравнению с «полноценными» процессами, и на фоне этого сокращение затрат на их взаимодействие еще более заметно и актуально.

11.3.1 Мьютексы – pthread mutex

Мьютекс (***Mutex***, ***MU***table ***EX***clusion) – практически не отличается от традиционного.

Объект ***pthread_mutex*** создается в пользовательском адресном пространстве процесса, доступен всем его потокам. Тип данных – **`pthread_mutex_t`**.

Три типа мьютексов, различающихся поведением:

– ***Fast*** (тип по умолчанию) – попытка повторного захват одним и тем же потоком приводит к блокировке точно так же, как и для любого другого потока, т.е. нет отдельной обработки рекурсивного захвата; возможна самоблокировка потока и общий ***тупик***

– ***Error checking*** – попытка повторного захвата обрабатывается как ошибка (код **`EDEADLK`**)

– ***Recoursive*** – разрешается повторный (рекурсивный) захват мьютекса одним и тем же потоком, на каждый «вложенный» захват потребуется дополнительное освобождение
(в Windows это единственный вариант поведения мьютексов)

Инициализация и разрушение объекта:

```
int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Вместо инициализации функцией можно использовать простое присваивание специального экземпляра-«инициализатора»:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Использование мьютекса – захват с блокирующим ожиданием в случае занятости, попытка захвата с ограничением времени ожидания, неблокирующая попытка захвата, освобождение:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(
  pthread_mutex_t *mutex,
  const struct timespec *abstime);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Свойства мьютекса управляются его ***атрибутами*** (в т.ч. тип мьютекса). Тип данных для описания атрибутов – **`pthread_mutex_attr_t`**, аналогично атрибутам потока используются функции/макросы доступа к отдельным атрибутам.

В типичном простейшем случае атрибуты мьютекса при его создании могут быть опущены (в функцию **`pthread_mutex_create()`** передается пустой указатель).

10.3.2 Барьеры – pthread_barrier

Особенность ***барьеров*** (***Barrier***) – счетчик синхронизируемых потоков: барьер «открывается» при достижении заданного числа потоков, ожидающих его открытия.

Объект ***pthread_barrier***, создается в «пользовательском» адресном пространстве процесса.
Тип данных **`pthread_barrier_t`**.

Инициализация и разрушение:

```
int pthread_barrier_init(
  pthread_barrier_t * barrier,
  const pthread_barrierattr_t *attr,
  unsigned count);
int pthread_barrier_destroy(
  pthread_barrier_t *barrier);
```

Использование – ожидание:

```
int pthread_barrier_wait(
  pthread_barrier_t *barrier );
```

11.3.3 «Циклическая блокировка» Spinlock

Синхронизация с «активным» ожиданием – внешне подобна мьютексу, но без перехода в состояние «Wait». Фактически соответствует циклу проверки состояния флага в ожидании его изменения. Очевидно, при использовании циклической блокировки процессор остается загружен и не может переключаться на другие задачи. Поэтому циклические блокировки имеют смысл только на очень короткое время, в течение которого загрузка процессора не критична. Но зато для таких блокировок выход и выход происходят тоже максимально быстро.

Предполагается, что переменная должна и будет меняться «извне» – другим потоком (выполняющимся на другом ядре) или обработчиком исключения.

Циклические блокировки используются внутри ядра системы как наиболее низкоуровневые, реализованные непосредственно средствами процессора примитивы синхронизации.

Для «прикладных» потоков предоставляется высокоуровневый объект синхронизации «циклическая блокировка» – ***pthread_spinlock***. Объект создается в «пользовательском» адресном пространстве процесса.

Идентификатор объекта – тип **`pthread_spinlock_t`**.

Инициализация и разрушение объекта:

```
int pthread_spin_init(
  pthread_spinlock_t *lock, int pshared );
int pthread_spin_destroy(
  pthread_spinlock_t *lock );
```

Использование объекта – примитивы в целом аналогичны применяемым к мьютексам, различается только реализация ожидания.

Захват с предварительным ожиданием освобождения:

```
int pthread_spin_lock(pthread_spinlock_t *lock);
```

Попытка захвата без ожидания (ошибка, если занято):

```
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Освобождение:

```
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

11.3.4 Futex

«***Фьютексы***» (***Futex***, ***F***ast ***U***niversal mut***EX***, также ***FU***ss) – альтернатива мьютексам, разнообразные варианты поведения. Реализуются на основе целочисленного счетчика в «пользовательском» адресном пространстве процесса.

```
int futex( int* uaddr, int op, …)
```

Постоянные параметры вызова – счетчик (по указателю) и код выполняемой операции, остальные зависят от операции.

Примеры операций:

– **FUTEX_WAIT**
– **FUTEX_WAKE**
– **FUTEX_REQUEUE**
– **FUTEX_CMP_REQUEUE**

и т.д.